VMM Verification Methodology Manual 活用テクニック

デバイスの記事

赤星博輝

第1回 VMMの概要とvmm_logの使いかた

Verilog HDLと比べると，SystemVerilogは検証機能が大幅に強化されている．しかし，これらの機能を我流で使ってしまうと，作成した検証用の部品が再利用しにくくなったり，SystemVerilogの能力を十分に引き出せない場合がある．こうした問題を軽減するために，検証ライブラリとその利用ガイドラインである“Verification Methodology Manual for SystemVerilog（VMM）”が提供されている．本連載では，VMMを活用するために必要なSystemVerilogの構文や記述例を紹介していく．（編集部）

2005年にSystemVerilog 3.1aがIEEE 1800として承認されました．現時点で多くのツール・ベンダがサポートすることを宣言しており，設計および検証言語としてのSystemVerilogから目が離せない状況になりそうです．

これまで，ハードウェアの検証言語としては，PSL（Property Specification Language）やOpenVera，e言語といった複数の言語がありました．しかし，ツールのサポートがばらばらで，別の設計環境に持っていくことが難しい場合がありました．標準化されたSystemVerilogについては，今後，早い段階で多くの対応ツールの出荷が期待されており，これまで乱立していた検証言語を統一してくれるのではないか，という希望もあります．そうした中でユーザが正しい選択を行うことが重要です．

RTL（register transfer level）設計ではVerilog HDLとVHDLの二つの標準言語が存在します．これらの言語が登場したころはこれで良かったのかもしれませんが，現時点ではデメリットのほうが大きいように感じます．今後，複数の検証言語が残ると，むだな投資が増えるように思います．

SystemVerilogは設計だけでなく検証についても多くの機能が追加されており，例えば制約付きランダム・パターン生成（以下，ランダム生成）やアサーション，機能カバレッジ，同期機構，連想配列，動的な配列，キュー（queue）などの機能を備えています．このような機能は，これまで，ハードウェアの世界では検証言語が備えていたのですが，SystemVerilogによって設計も検証も統一的に扱えるようになりました．設計も検証も同じ言語を使用することで，習得のためのコストが低くなると期待されています．



写真1 Verification Methodology Manual for System Verilogとその邦訳

● 検証スペシャリストの“技”を学ぼう

それでは，SystemVerilogを使えばだれでも検証を効率良く行うことができるのでしょうか？ その答えは，「ある程度まではできる」と言えますが，どこまでできるかは人やグループによって決まると言えます．そこで“Verification Methodology Manual for SystemVerilog”（通称VMM，邦訳「ベリフィケーション・メソドロジ・マニュアル」，CQ

KeyWord: SystemVerilog，VMM，検証ライブラリ，検証IP，クラス，プロパティ，メソッド，継承，オーバロード，スーパクラス，vmm_log，VCS

出版社 刊，**写真1**）の登場となります．

VMMはこれまでいろいろな検証を行ってきたスペシャリストが経験を踏まえて作成した検証ライブラリとその検証のやりかたを記述したものです．VMMを読むことで，検証のスペシャリストが実際にやっている方法を知ることができます．また，それが単なる読みものではなく，実際にSystemVerilogシミュレータ（米国Synopsys社の「VCS」など）を使って実行できることが大きな利点です．読むだけでなく，それを実際に使用することで，本書の著者であるJanick Bergeron氏のような検証スペシャリストのやりかたを理解しやすくなります．

VMMは，どのようなねらいで作られたのでしょう．筆者の勝手な解釈をもとに，以下で説明していきます．

1）検証における再利用を促進する

どうしても業務では設計に比重が置かれており，検証にかけられる工数が限られている設計グループが多いと思います．このとき，単体モジュールの検証，複数モジュールの検証，ソフトウェアとハードウェアを合わせた検証などを効率良く行えているでしょうか？ また，前回の検証で使った部品を再利用している例はどのくらいありますか？

VMMでは再利用を促進するためにライブラリを用意しています．そのライブラリでは，通信用のチャネル，データ用のクラス，ログを管理するメッセージ・サービス，ランダム生成用のクラスなどがあります．例えばテストベンチの中に複数の部品があり，それらが通信する場合，個別の部品の再利用性は通信方法に依存します．VMMはこの通信方法を統一しているため，再利用性が向上します．

設計ではIP（intellectual property）コアを活用することがポイントになっていますが，検証でもIPが重要になってきています．そのときに，勝手なルールで作られたIPでは，それを使うだけでひと苦労です．VMMでは用意されたライブラリを使い，さらにその使用方法もルール化されています．外部から検証IPなどを導入しやすくなります．

2）検証範囲を広げる

現在のハードウェアは規模が大きくなり，実現する機能が複雑化しています．これは，検証時に探索する空間が広くなっていることを意味します．設計者や検証エンジニアがすべての信号の挙動を指定しなければならないダイレクト・テストでは，その広くなった空間を検証するすべてのシミュレーション・パターンを作ることは困難です（例えば，32ビットの加算をチェックするパターンは2^{64}パターン必要）．SystemVerilogはランダム生成に対応しています．VMMの考えかたでは，このランダム生成の機能を活用し，ダイレクト・テストよりも広い範囲を検証することが前提となっています．

「ランダム生成では，意味のないパターンが生成されてしまう」といって嫌われる場合もあります．しかし，SystemVerilogではランダム生成に制約を付けることができ，ありえないパターンを除外した有効なランダム値だけを発生します．これにより，短い記述で，多くのシミュレーション・パターンを生成できます．と言っても，VMMがダイレクト・テストを否定しているわけではありません．ダイレクト・テストはランダム生成の一つのバリエーションとして実現できるようになっています．そのため，ランダム生成では目標とする検証の範囲に到達できない場合，ダイレクト・テストに簡単に切り替えることができます．

3）検証のチェックを自動化する

ダイレクト・テストを行っている場合には，出力を人手（による目視）で確認してもなんとかなりました．1分間に数個のシミュレーション・パターンを作成した場合，数個の出力を数分でチェックできれば，その検証作業はなんとか実施可能と言えると思います．しかし，ランダム生成を使うと，コンピュータが山のようにシミュレーション・パターンを生成してくれます．あっという間に数万パターンを作り出しますが，その出力を人手で確認できるのでしょうか？ この場合，その検証作業は実施が困難です．出力をチェックしないシミュレーションに意味はありません．

VMMでは，入力に対してその応答をチェックするためにアサーションやスコアボードを使うことが前提となっており，それらの使いかたについて説明されています．

4）検証の終わりを判断する

ダイレクト・テストで検証を行っている場合は，すべてのテストベンチで問題が発生しなければ検証の終わりと言えますが，ランダム生成の場合はいつが検証の終わりなのでしょう？ ランダム生成を使っている場合，カバレッジをその判断材料にすることになります．一般的にはコード・カバレッジがよく使用されていますが，コード・カバレッジだけでは不十分な点があり，機能カバレッジも使用します．機能カバレッジを収集するため，SystemVerilogでは`covergroup`や`cover property`といった構文が追加されています．

● 人に対する教育やトレーニングが必要に

「これからは検証がたいへんになる」とは言いながら，検証に対してなかなか積極的に投資できない，という問題があります．投資と言っても，単にツールを買うだけでなく，人に対する教育やトレーニングが必要です（ツールを買ったけどうまく使えないという例は非常に多い）．

検証言語を導入する場合には，教育のための期間がかならず必要になります．この部分のコストや時間をどのように確保するのかも含めて考えなければなりません．

● VMMとは？

先述のようにVMMとは，検証ライブラリとその検証ライブラリを使った検証環境の構築のしかたと言えます．

検証ライブラリはクラスという形でまとまっており，データを保持するvmm_data，データを渡す通信路であるvmm_channel，処理を記述するvmm_xactor，メッセージを出力するvmm_log，ランダム生成を行うvmm_atomic_genとvmm_scenario_gen，検証環境をまとめるvmm_envといったクラスがあります．SystemVerilogはオブジェクト指向言語なので，これらのクラスをもとに必要な機能を持ったクラスを作成します．

簡単な例を**図1**で説明します．top階層に検証対象の回路（DUT：device under test）と検証のクラスvmm_envから検証環境のクラスがインスタンスされます．vmm_atomic_genからランダム生成を行うクラスを作り，vmm_xactorからドライバやモニタ，スコアボードといった部品を作り，それらの間で通信するためにはvmm_channelを利用します．さらにVMMでは，このvmm_envから作ったクラスの中で実行フローを制御します．**図1**に示したタスク呼び出しはVMMによって決められており，それぞれで実行することも決められています．例えば，buildでは検証に使用するクラスをインスタンスし，startではインスタンスした部品を起動します．このようなやりかたがVMMには記載されています．

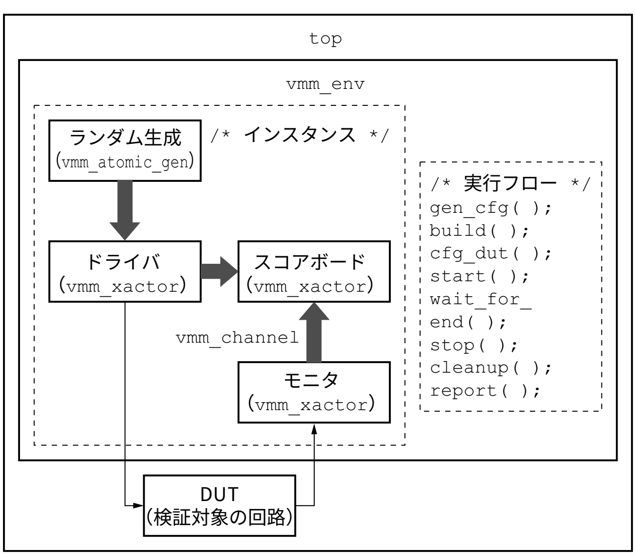


図1　VMMでの検証環境例

検証用ライブラリを使用することで，効率良く設計できる．また，ルールによって実行フローなどが指定されており，標準的な検証環境を構築できる

これらのクラスを使って検証環境を作成することで，以下の利点があります．

- 少ないコードで多くのテストを実行できる
- テストに依存するコードを少なくできる
- 再利用が促進される

とくに再利用については，設計グループ内の再利用にとどまらず，外部も含めた再利用を行えることが重要な点です．VMMに従った検証IPも同じクラスをもとに作成されているため，日ごろテストベンチなどで書いているルールをそのまま使用できることになります．VMMそのものが標準化されているため，こうした利点が出てきます．

本連載では，VMMの活用に必要なSystemVerilogの構文や記述例を紹介します．今回は，VMMのライブラリの一つであるvmm_logの使いかたを説明します．

● まず，SystemVerilogのclassを理解する

VMMでは，クラス単位に検証の部品を作ります．以下ではSystemVerilogのclassについて説明します．JavaやC++などを知っている方であれば，だいたい同じような働きをすると考えていただいて問題ありません．これらについて，少し解説していきます．

SystemVerilogのclassでは，データを保持するプロパティ（変数）とそのclassのプロパティに対する処理を行

リスト1　2科目のデータを保持し，その平均値や表示を行うクラス（grade）

```
class grade;
  integer math; //数学
  integer eng;  //英語
  function integer average( );
    average=(math+eng)/2;
    $display("average=%3d",average);
  endfunction
  task show(string str);
    $display("%s math=%3d, english=%3d",str,math, eng);
  endtask
endclass
```

うメソッド(functionやtask)を定義できるようになりました．例えば，2科目のデータを保持し，その平均値や表示を行うクラスgradeを**リスト1**に示します．

1) クラス変数とハンドル

クラスを定義したら，そのクラス変数を宣言します．

```
grade   smith;
```

ここで注意が必要です．このクラス変数というのは，ハンドル(C言語で言うポインタ)のみを保持することができます．そのため，クラス変数そのものは数学や英語の点を保存するデータ領域を持っていません．どこかに領域を確保した場合に，その場所(ハンドル)だけを記憶します．最初は，クラス変数はどこの場所も指しておらず，初期値にはnullが入っています．

2) クラス・インスタンスの生成

実際に数学や英語の点を保存するデータ領域を確保するにはnewを使用します．

```
smith = new( );
```

とすることで，データを保存する領域が確保されました．

3) クラスのプロパティへの代入・参照

領域を確保したら，値を代入できます．Verilog HDLでは階層を示すときに . を使用しましたが，クラス変数のプロパティにアクセスするときも . を使用します．クラス変数smithのmathに100を書き込む記述を示します．

```
smith.math = 100;
```

読み出すだけでなく書き込むときにも，同じように記述できます．英語の点数が数学より20点低い場合には，以下のように記述します．

```
smith.eng = smith.math-20;
```

4) クラスのメソッドを使用する

最初にクラスを定義したときに，プロパティと同時にメソッドを定義しました．このメソッドを呼び出すときにも .を使って呼び出します．クラス変数smithに対してshowとaverageというメソッドを呼び出すには，以下の記述となります．

```
smith.show("smith:");
smith.average( );
```

5) クラス変数の代入

SystemVerilogで最初につまずくのはこのクラス変数の代入だと思います．**図2**を使って説明していきます．二つのクラス変数を宣言し，同時にインスタンスを作成します．

```
grade smith = new( );
grade tom   = new( );
```

このとき，以下のような代入を行ったら何が起こるのでしょうか？

```
tom = smith;
```

それぞれのインスタンスの内容をコピーするのでしょうか？ 残念ながら違います．クラス変数の代入は値が代入されるわけではなく，ハンドルが代入されます．そのため，**図2**に示すように，代入した後からクラス変数smithとtomが同じ領域を指すことになります．tom.math=60;を実行し，smith.show( ) と tom.show( )が同じ結果になることで，それが正しいことを示しています．

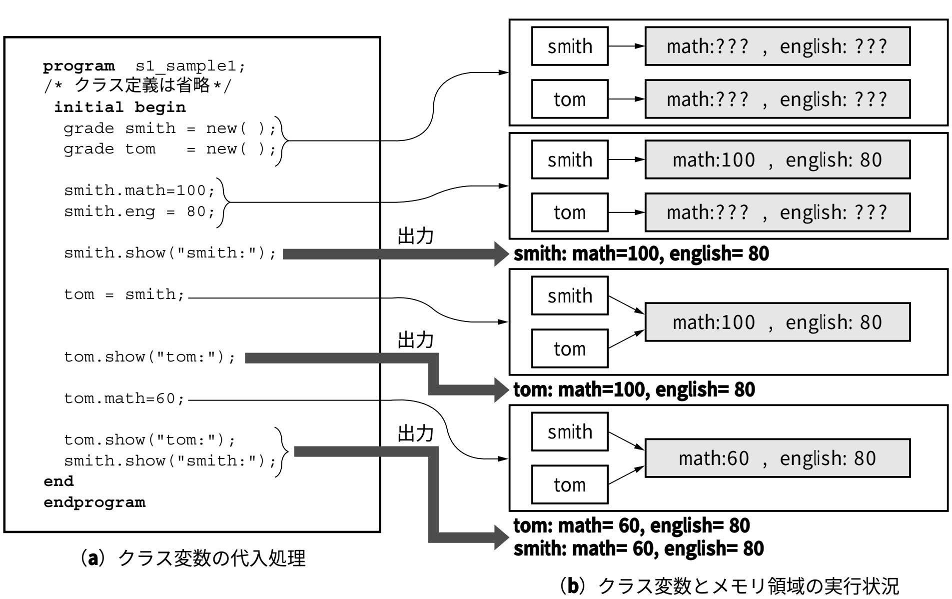


図2
SystemVerilogのクラス
クラス変数はハンドル(ポインタ)を格納するだけである．代入した場合の動きに注意してほしい

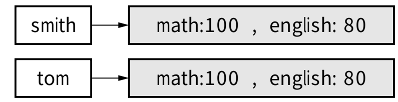
```
class grade;
  /* 省略 */
  function void  copy(grade dst);
    dst.math= math;
    dst.eng = eng;
  endfunction
endclass
```

（**a**）追加する copy メソッド

```
initial begin
  grade  smith = new( );
  grade  tom   = new( );

  smith.math=100;
  smith.eng = 80;
  smith.copy(tom);
   ・・・
end
```

（**b**）copy メソッドを使った記述



（**c**）smith の内容を tom にコピーする

図3　クラス変数が指す内容のコピー
インスタンスの内容をコピーしたい場合には，プロパティごとの代入が必要になる．めんどうなので，copy メソッドを作ることが一般的

リスト2　継承を使って新しいクラスを宣言
継承を使えば，差分だけを記述することで，新しい機能を持ったクラスを作成できる

```
class grade3 extends grade;

  integer chem; // 化学

  function integer average( );
    average=(math+eng+chem)/3;
    $display("average=%3d",average);
  endfunction

  task show(string str);
   $display("%s math=%3d, english=%3d,chemistry=%3d",str,math, eng,chem);
  endtask
endclass
```

gradeを継承してクラスgrade3を定義
新しいデータのchemのプロパティ
新しいデータが増えたので，averageのfunctionも変更する
新しいデータが増えたので，出力するshowも変更する

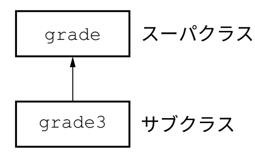


図4　継承したときの呼びかた
元となったクラスをスーパクラス（親クラス），元のクラスを継承して作ったクラスをサブクラス（子クラス）と呼ぶ

もし，インスタンスの値をコピーしたい場合には，クラス・プロパティに対して操作を行う必要があります．ただし，クラス・プロパティに対する記述は長くなるので，**図3**のような copy メソッドを用意することがよく行われます．

6）クラスの継承

SystemVerilog はオブジェクト指向言語なので，あるクラスをもとに，新しいクラスを作ることができます．これにより，用途によって毎回クラスを一から作る必要がなくなり，差分を作るだけで済みます．

先ほどのクラスに化学の成績を持つ chem というプロパティを加えるために，継承という手法を使います．**リスト2**のようにクラス宣言に「extends　元のクラス」を付けることで，元のクラスのプロパティやメソッドを引き継いだ新しいクラスを作成できます．このとき新しく作ったクラスをサブクラス，元になったクラスをスーパクラスと呼びます（**図4**）．

7）クラス・メンバのオーバロード

継承したクラスでは，メソッドなどを変更することが可能です．平均値を計算するメソッド average と，出力するメソッド show はともに数学と英語しか見ていないので，そのままでは役に立ちません．このとき，スーパクラスのメソッドを上書き（オーバロード）することができます．これによって，数学と英語，化学の成績を表示したり，平均を求めたりできるようになります．

8）スーパクラスへの代入

新しいクラスを作るよりクラスの継承を使ったほうがよい理由は，「差分を作るだけで済むから」というだけではありません．継承して作ったクラスのハンドルをスーパクラスに代入できることも挙げられます．

これがなぜうれしいかというと，新しい機能を持つクラスに変えても，そのほかの部分は変更しなくてもよいからです．**図5**の記述では，最終的に grade の配列 student でメソッドなどを呼んで処理するのですが，新しい grade3 のクラスのインスタンスでも代入できるのです．**リスト3**にその例を示します．

例えば，異なったシミュレーション・パターンを生成する新しいクラスを継承によって作っておけば，テストベンチの構造は変更不要ということになります．ただし，このときにクラスのメソッドに virtual を指定するのがポイントです．クラス grade のメソッドに，**リスト4**のように virtual を付けてみます．

この virtual を付けることで，**図5（c）**の呼び出しのときに本来のクラスのメソッドを呼び出せます．virtual を

```
grade  tom   = new;
grade3 marry = new( );
grade  student[2];
student[0] = tom;
student[1] = marry;
・・・
student[0].show("tom:");
student[1].show("marry:");
```

(a) スーパクラスをベースクラスに代入できる

```
tom: math=100, english= 80
marry: math= 50, english= 60
```

virtualがないとstudentのクラスのshowを呼ぶ

(b) メソッドをvirtual宣言しない場合の実行結果

```
tom: math=100, english= 80
marry: math= 50, english= 60,chemistry     70
```

virtualで宣言された場合は，student[1]はmarryのハンドルを持っているので，grade3のshowを呼び出す

(c) メソッドをvirtual宣言した場合の実行結果

図5　メソッドにはvirtual

virtualを付けてメソッドを定義すると，オブジェクト指向言語の威力を最大限に活用できる

リスト3　新しいgrade3のクラスのインスタンスでも代入できる

```
grade  tom   = new;
grade3 marry = new( );
grade  student[2];
student[0] = tom;
student[1] = marry;
```

リスト4　クラスgradeのメソッドにvirtualを付ける

```
class grade;
  integer math; //数学
  integer eng;  //英語
  virtual function integer average( );
    average=(math+eng)/2;
    $display("average=%3d",average);
  endfunction
  virtual task show(string str);
    $display("%s math=%3d, english=%3d",str,math,eng);
  endtask
endclass
```

付けないと**図5(b)**からわかるようにスーパクラスのメソッドを呼ぶことに注意してください．VMMでは，この継承を使うことにより，非常に少ない変更で異なったシミュレーション・パターンを生成することが可能となります．

● レポートやエラー出力のためのライブラリvmm_log

VMMでは，さまざまな部品で発生するレポートやエラーなどを出力するためのライブラリとして，vmm_logを使用します．出力するメッセージに関するサービスを提供しているので，「メッセージ・サービス」と呼びます．VMMでは，このvmm_logが使えないとなにもできません．

検証ではメッセージを出力し，デバッグや解析に利用します．しかし，その出しかたなどにいろいろと問題があります．人によってメッセージの表示のしかたが異なったり，メッセージの出力を止められなかったりします．それらを全部解決しようというのがvmm_logです．ここではvmm_logの主要な部分について紹介します．

vmm_logは，以下のサービスを提供します．

- 表示するメッセージの制御
- 複数のメッセージ・サービスのインスタンス
- メッセージ・サービスの階層化
- それぞれのメッセージ・サービスの制御，およびプロモーション/デモーション
- レポート機能

● エラー・メッセージを区別しよう

vmm_logのサービスを活用するためには，まずメッセージがどのような種類のものであるかを分類します．例えば，デバッグのための情報，エラーに関する情報，エラーではないがワーニング(警告)として出したい情報，どう動いたかを知るトレースの情報などです．VMMでは**表1**に示す種類が用意されており，そのために使用するマクロが用意されています．ここでメッセージをきちんと分類しておくと，あとでVMMの機能を利用しやすくなります．

VMMが提供するメッセージ・サービスを使用した簡単な例を**リスト5**に示します．今回はprogramというキーワードを使っていますが，これは検証モジュールと考えてください．programを使うことで，テストベンチと設計の間のレーシングが発生しないなどの利点があり，SystemVerilogでは検証モジュールにこのprogramを使うことが推奨されています．

表1　メッセージの分類と使用するマクロ

メッセージには，エラー出力やトレース出力などの異なった役割があるはず．きちんと分類しておくと，多くのメリットが出てくる

メッセージの種類	使用するマクロ	エラーのレベル
verboseメッセージ	`vmm_verbose	VERBOSE_SEV
debugメッセージ	`vmm_debug	DEBUG_SEV
traceメッセージ	`vmm_trace	TRACE_SEV
noteメッセージ	`vmm_note	NORMAL_SEV
warningメッセージ	`vmm_warning	WARNING_SEV
errorメッセージ	`vmm_error	ERROR_SEV
fatalメッセージ	`vmm_fatal	FATAL_SEV

リスト5
エラー・メッセージを出力する記述
メッセージを出力するときは，表1のマクロを使う

```
program  s1_sample6;
`include "vmm.sv"

vmm_log  ms1;

initial begin
  ms1 = new("program", "MS1");
  `vmm_verbose(ms1, "verbose");
  `vmm_debug    (ms1, "debug");
  `vmm_trace    (ms1, "trace");
  `vmm_note      (ms1, "note");
  `vmm_warning(ms1, "warning");
  `vmm_error    (ms1, "error");
  `vmm_fata     (ms1, "fatal");
end

endprogram
```

VMMを使うためにはvmm.svのファイルをインクルードします．VCS-2005.06-SP1（Synopsys社のシミュレータ）では，このファイルはインストールされたディレクトリのetc/rvmにあります．

vmm_logクラス変数の定義は，Verilog HDLの信号を定義するのと同じです．ms1というクラス変数の宣言は，以下の記述になります．

```
vmm_log  ms1;
```

上述したように，宣言しただけではメッセージ・サービスとしては使えないので，インスタンスを生成します．これはVerilog HDLでモジュールを定義するだけでなく，インスタンスするのと同じようなものです．

インスタンスの生成時に，logを使用するクラスなどの名まえとインスタンスの名まえをnewに与えて作成します．この名まえは自由に決めてよいのですが，メッセージの出力を制御するときに使用するので，vmm_logを複数インスタンスしたときに明確に区別できるようになっていることが必要です．

今回は，programのメッセージ・サービス1という意味で，MS1という名まえでインスタンスを作成しました．

```
ms1 = new("program", "MS1");
```

メッセージを出力するためには，**表1**のマクロを使って第1引き数にクラス変数を，第2引き数に出力したいメッセージを指定します．例えば，ワーニングのメッセージを出力したい場合には，以下のように記述します．

```
`vmm_warning(ms1, "warning");
```

注意点は，`vmm_fatalを呼び出すとその時点でシミュレーションが停止することです．ほかのマクロはシミュレーションを継続して実行します．

それではこの記述をコンパイルし，実行してみます．

```
% vcs  -sverilog  +incdir+$VCS_HOME/etc/rvm ファイル名
% ./simv
```

と入力すると，一部のメッセージのみが出力されます．デフォルトではnote，warning，error，fatalに関するメッセージが出力されます．ここで，なぜメッセージが全部出てこないのかと言うと，どのメッセージを出力するかをユーザが変更できるからです．そのほかのメッセージを出力するためには，実行時オプションで出力レベルを変更する方法と，記述中で出力レベルを変更する方法があります．これにより，ほんとうに必要な情報だけを取り出すことができます（むだな情報を出力すると処理が遅くなる）．

● メッセージの出力レベルを切り替える

まず，実行時オプションで出力レベルを変更してみましょう．このとき，オプションは +vmm_log_default= を使用し，これに続いてverbose，debug，trace，warning，error，fatal，normalを指定します．例えば，すべてのメッセージを出力させたい場合には，以下のように入力します．

```
% ./simv +vmm_log_default=verbose
```

オプションによって，どのレベルのメッセージが出力されるかを**表2**に示します．

複数のメッセージ・サービスをインスタンスし，記述による出力レベルの変更を行った記述が**リスト6**になります．二つのメッセージ・サービスMS1，MS2を作成し，TRACEレベルのメッセージを出力しています．ただし，TRACEなのでデフォルトで実行した場合にはこれらのメッセージは出力されません．先ほどの実行時オプションで出力レベルを変更したり，より細かい指定を行えるset_verboseメソッドを次に紹介します．

リスト6のAのコメント（//）を解除し，コンパイルと実行を行うと，すべてのメッセージが出力されます．

```
ms1.set_verbosity(vmm_log::
          TRACE_SEV,"/./", "/./"); // A
```

これは，logの名まえとインスタンスの名まえが任意のメッセージ・サービスの出力レベルをTRACEに設定します（どのレベルでどのメッセージが出力されるかについては**表2**を参照）．

また，BのようにMS1のインスタンス名を持ったメッセ

表2
オプションで出力されるレベル
○が付いたメッセージが出力される

	normal	verbose	debug	trace	warning	error
`vmm_verbose		○				
`vmm_debug		○	○			
`vmm_trace		○	○	○		
`vmm_note	○	○	○	○		
`vmm_warning	○	○	○	○	○	
`vmm_error	○	○	○	○	○	○
`vmm_fatal	○	○	○	○	○	○

リスト6
set_verbosityを使った出力レベル制御
複数のメッセージ・サービスを使用し，個別に制御するには，set_verbosityを使用する

```
program s1_sample7;
`include "vmm.sv"

initial begin
  vmm_log ms1 =new("program", "MS1");
  vmm_log ms2 =new("program", "MS2");

  //ms1.set_verbosity(vmm_log::TRACE_SEV,"/./", "/./"); // A
  //ms1.set_verbosity(vmm_log::TRACE_SEV,"/./", "/MS1/"); // B
  //ms1.set_verbosity(vmm_log::TRACE_SEV,"/program/”, "/./"); // C
  `vmm_trace  (ms1, "trace1");
  `vmm_trace  (ms2, "trace2");
end

endprogram
```

ージ・サービスだけに対して出力レベルを変更することもできます．

```
ms1.set_verbosity(vmm_log::
          TRACE_SEV,"/./", "/MS1/"); // B
```

さらに，logの名まえでメッセージ・サービスを指定して，出力レベルを変更することもできます．

```
ms1.set_verbosity(vmm_log::
      TRACE_SEV,"/program/", "/./"); // C
```

確かに個別のメッセージ・サービスの出力レベルを制御できることは便利なのですが，制御する数が多くなってくると取り扱いがめんどうです．例えば，全部で100個あるメッセージ・サービスのインスタンスに対して50個の出力レベルを変更しようとしたらたいへんです．そのため，vmm_logでは階層構造を作れるようになっています．メッセージ・サービスで階層構造を作った例が**リスト7**です．

メッセージ・サービスのインスタンスを生成するときに上位のメッセージ・サービスがある場合は，第3引き数として与えることで階層を作成できます．第3引き数はデフォルト引き数としてnullが与えられているので，階層がない場合には与える必要がありません．以下の記述では，ms2がms1の下位階層となります．

```
ms2 = new("LOG",  "MS2", ms1);
```

ある階層以下のメッセージ・サービスに対して出力レベルを変更するためには，set_verbosityの第3引き数に

リスト7　階層を持ったメッセージ・サービス
メッセージ・サービスに階層構造を持たせることにより，例えばある階層以下のメッセージをまとめて制御できる

```
program s1_sample8;
`include "vmm.sv"

vmm_log  ms1,ms2,ms3;

initial begin
  ms1 = new("LOG",  "MS1");
  ms2 = new("LOG",  "MS2", ms1);
  ms3 = new("LOG",  "MS3", ms2);

  ms1.set_verbosity(vmm_log::TRACE_SEV, "/./",
                              "/MS1/",1); // D
  `vmm_trace  (ms1, "trace");
  `vmm_trace  (ms2, "trace2");
  `vmm_trace  (ms3, "trace3");
end
```

‘1’を設定します(これもデフォルト引き数が与えられており，指定しない場合には‘0’となっている)．

```
ms1.set_verbosity(vmm_log::TRACE_SEV,
                  "/./", "/MS1/",1); // D
```

また，メッセージ・サービスの階層を確認するために，listを使ってそのイメージをテキストで出力できます．

```
ms1.list( );
```

これは，実際に作成した階層が正しいかどうかを確認する際に使用できます．

VMMのメッセージ・サービスを使う利点は，生成したメッセージに対する情報を自動的に収集してくれることにあります．検証ではエラー数やワーニング数が重要ですが，

リスト8
レポートの生成
vmm_log を使うと，エラー・ワーニングに関する情報を自動的に収集してくれる

```
program s1_sample9;
`include "vmm.sv"

vmm_log  ms1,ms2,ms3;

initial begin
  ms1 = new("LOG",  "MS1");
  ms2 = new("LOG",  "MS2",ms1);
  ms3 = new("LOG",  "MS3",ms2);

  `vmm_trace  (ms1, "trace");
  `vmm_error  (ms1, "error");
  `vmm_trace  (ms2, "trace2");
  `vmm_error  (ms2, "error2");
  `vmm_trace  (ms3, "trace3");
  `vmm_warning(ms3, "warning3");

  ms1.report("/./", "MS1");
  ms1.report("/./", "MS1",1);
  ms1.report("/./", "MS2");
end

endprogram
```

リスト9　メッセージのプロモーションとデモーション
あとから，メッセージの出力レベルを変更できる

```
program s1_sample10;
`include "vmm.sv"

vmm_log ms1=new("program", "MS1");

initial begin
  ms1.modify("/./","/./",0,
    /*TYPE*/,vmm_log::DEBUG_SEV ,/*TEXT*/ ,
    /*NEW TYPE*/,        vmm_log::WARNING_SEV,/*HANDLING*/);
  ms1.modify("/./","/./",0,
    /*TYPE*/,vmm_log::WARNING_SEV ,/*TEXT*/ ,
    /*NEW TYPE*/,        vmm_log::NORMAL_SEV, /*HANDLING*/);

  `vmm_debug(ms1, "debug");
  `vmm_warning(ms1, "warning");

  ms1.report("/./","/MS1/" );
end

endprogram
```

それをレポートする機能があらかじめ作り込まれています．わざわざ設計者や検証エンジニアが作る必要はなく，**リスト8**のようにreportを呼びだすだけで出力されます．このとき，どのメッセージ・サービスのレポートを生成するのかを指定できるようになっていて，例えばインスタンス名がMS1だけのレポートを表示する場合は，以下のようにします．

```
ms1.report("/./", "/MS1/");
```

また，MS1からその下位階層を含めてレポートを表示したい場合には，以下のように第3引き数に‘1’を与えます．

```
ms1.report("/./", "/MS1/",1);
```

● メッセージの出力レベルを後から変更できる

VMMのメッセージ・サービスのもう一つの機能として，メッセージのレベルを後から変更できることがあります．これはメッセージのプロモーション/デモーションと呼ばれています．例えば，DEBUGレベルのメッセージをWARNINGにプロモーション（昇格）する記述は，以下のようになります．

```
ms1.modify("/./","/./",0,
/*TYPE*/,vmm_log::DEBUG_SEV ,/*TEXT*/ ,
/*NEW TYPE*/, vmm_log::WARNING_SEV,
                    /*HANDLING*/);
```

また，WARNINGレベルのメッセージをNORMALにデモーション（降格）する記述も同じように記述できます．

```
ms1.modify("/./","/./",0,
/*TYPE*/,vmm_log::WARNING_SEV ,/*TEXT*/ ,
/*NEW TYPE*/,vmm_log::NORMAL_SEV,
                    /*HANDLING*/);
```

リスト9の記述を実行してみると，メッセージがレベルに応じて変更されています．また，このレポートの結果が重要です．VMMのライブラリを使っていて「よくできているなぁ」と関心するのですが，このように出力レベルを変更したとき，例えばエラーやワーニングのレベルを下げる（デモーションする）と，別にカウントされることになっています．今回のレポート結果を以下に示します．ワーニングがデモーションされたと出力されています．

```
Simulation PASSED on /./ (/MS1/) at 0 (0 warnings,
        0 demoted errors & 1 demoted warnings)
```

これにより，ケアレス・ミスによってmodifyを使ったりすることを防止できます．意図的にエラーを隠そうとすることも防止できます．

VMMのライブラリはいろいろなケースに対応できるようになっており，自分たちで作るよりも高機能な検証用の部品がすでに使える状態になっている点が大きな利点だと思います．次回は，制約付きランダム・パターン生成の機能を中心に紹介します．

あかぼし・ひろき　（株）ロジック・リサーチ

<筆者プロフィール>
赤星博輝．ハードウェアからソフトウェアまで担当するなんでも屋．現在はディジタル検証ではVerification Methodology ManualとSystemVerilogを推進し，ハードウェア検証のためにリアルタイムOS技術にも進出．